

(11)Publication number : **2000-250758**  
(43)Date of publication of application : **14.09.2000**

**G06F 9/445**

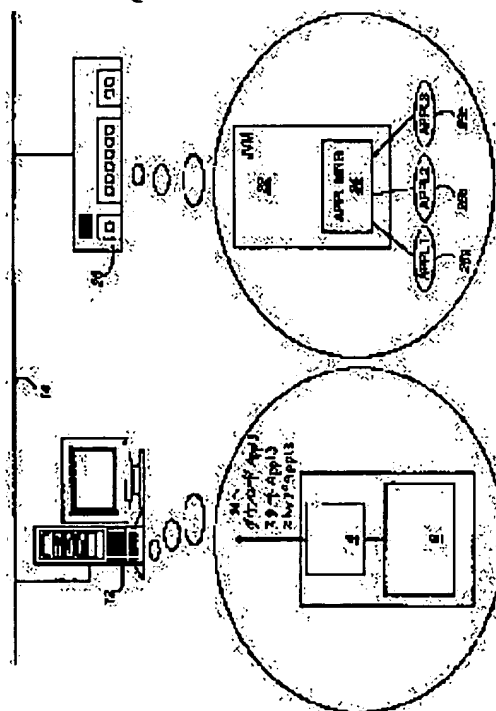
(71)Applicant : **HEWLETT PACKARD CO <HP>**

(72)Inventor : **FRANK P JUDGE**  
**CHIA-CHU DORAND**

Priority number : 99 259616    Priority date : 01.03.1999    Priority country : US

(57)Abstract:

**SOLUTION:** Integrated equipment 20 is provided with a Java (R) virtual machine(JVM) 22, a memory having an application cache and data cache and a network interface. The JVM 22 is provided with a class loader and an execution unit and the class loader is executed while using an application manager 24. The application manager 24 is a non-general network base class loader and loads a class file from a network 14 for each request. When the request of downloading a class file is received by a network protocol such as a server socket, the application manager 24 receives and analyzes the class file from the network 14 and that class file is made into class objects 28a and 28b.



**BEST AVAILABLE COPY**

[Date of final disposal for application]

[Patent number]

[Date of registration]

[Number of appeal against examiner's decision of rejection]

[Date of requesting appeal against examiner's decision of rejection]

[Date of extinction of right]

Copyright (C); 1998,2003 Japan Patent Office

(19) 日本国特許庁 (J P)

(12) 公開特許公報 (A)

(11) 特許出願公開番号

特開2000-250758

(P2000-250758A)

(43) 公開日 平成12年9月14日 (2000.9.14)

(51) Int. CL.  
G 0 6 F 9/445

識別記号

F I  
G 0 6 F 9/06

テーマコード(参考)

4 2 0 J

審査請求 未請求 請求項の数 1 O L (全 18 頁)

(21) 出願番号 特願2000-55728(P2000-55728)

(22) 出願日 平成12年3月1日(2000.3.1)

(31) 優先権主張番号 2 5 9 6 1 6

(32) 優先日 平成11年3月1日(1999.3.1)

(33) 優先権主張国 米国 (US)

(71) 出願人 399038580

ヒューレット・パッカード・カンパニー

HEWLETT-PACKARD COMPANY

アメリカ合衆国カリフォルニア州パロアルト  
ハノーバー・ストリート 3000

(72) 発明者 フランク・ビー・ジャッジ

アメリカ合衆国コロラド州フォートコリンズ  
ホルヨーク・コート 525

(72) 発明者 チャーチュ・ドーランド

アメリカ合衆国コロラド州フォートコリンズ  
レバブリック・ドライブ 618

(74) 代理人 100078063

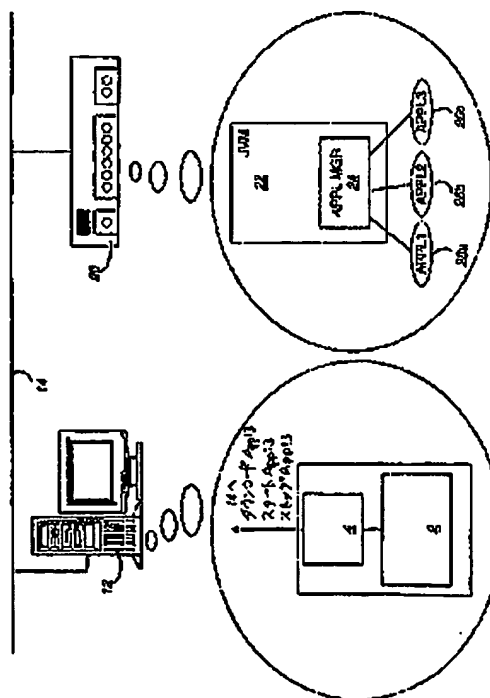
弁理士 上野 英夫

(54) 【発明の名称】 組込み機器用 Java アプリケーションマネージャ

(57) 【要約】

【課題】 組込み機器において、Java 仮想マシン等のハードウェア非依存型プロセッサが機器にインストールされることにより、アプリケーションが機器にダウンロードされその機器上で実行されることが可能になる。しかしながら、この Java 言語は、プログラムロードの実行もロードされたプログラムを管理する機能も提供していない。また、メモリの制約された組込み機器内でアプリケーションをダウンロードおよび管理する手段も提供していない。

【解決手段】 本発明の Java ベースのアプリケーションマネージャは、組込み機器にインストールされた Java 仮想マシン内で実行される Java アプリケーションのダウンロード、実行およびキャッシングを制御する。さらにアプリケーションの実行の開始、初期化および停止、およびメモリ管理のための機能を提供する。



## 【特許請求の範囲】

【請求項1】 組込み機器において1つ以上のアプリケーションを管理するアプリケーションマネージャであって、前記組込み機器が、クラスオブジェクトを格納するアプリケーションキャッシュと、ネットワークを介してクライアントと通信するネットワークインタフェースと、該組込み機器にインストールされ該組込み機器上で実行されるJava仮想マシンと、を備えるアプリケーションマネージャにおいて、(a) エントリを有し、前記エントリの各々が、前記組込み機器に現在ロードされているクラスオブジェクトを識別するクラスオブジェクトリストと、(b) 前記ネットワークインタフェースを介して前記クライアントから受信するアプリケーションクラスをロードし、前記アプリケーションクラスに対する新たなクラスオブジェクトを生成し、前記アプリケーションキャッシュに前記新たなクラスオブジェクトを格納し、前記新たなクラスオブジェクトを前記アプリケーションキャッシュにロードされているものとして識別するために、前記クラスオブジェクトリストに新たなエントリを付加するクラスローダメソッドと、を具備するアプリケーションマネージャ。

## 【発明の詳細な説明】

## 【0001】

【発明の属する技術分野】 本発明は、組込み機器においてアプリケーションを管理するアプリケーションマネージャに関する。

## 【0002】

【従来の技術】 組込み機器環境では、Java仮想マシン等のハードウェア非依存型プロセッサがしばしば機器にインストールされており、それによって、プログラムが機器にダウンロードされその機器上で実行されることが可能になっている。このようなシステムにより、Sun MicrosystemsによるJava（登録商標）等のハードウェア非依存型言語によって書かれたプログラムを、Java（登録商標）環境をサポートするあらゆるハードウェアにダウンロードし、特定の用途のためにカスタマイズすることができる。このカスタマイズは、しばしば機器の「パーソナリティ」と呼ばれる。機器のパーソナリティを定義するために、組込み機器内で複数のアプリケーションが協調して実行されている。このように、機器を、それぞれ一意に機能するよう動的に構成することができる。例えば、冷蔵庫等の組込み機器を、その中身を自動的に追跡するようカスタマイズすることができる。食品の種類および同種の食品の命名規則が文化によって異なるように、食品の種類および命名規則を記述する新たなパーソナリティをJava動作可能な組込み機器である冷蔵庫にダウンロードすることにより、その組込み機器の冷蔵庫を、ユーザの特定の文化に対してカスタマイズすることができる。

## 【0003】

【発明が解決しようとする課題】 組込み機器ドメインでは、Java（登録商標）仮想マシン等のハードウェア非依存型プロセッサが機器にインストールされることにより、アプリケーションが機器にダウンロードされその機器上で実行されることが可能になる。Java（登録商標）言語は、新たなプログラムを動的にロードするようプログラムを構築する基本的な機能を提供する。しかしながら、このJava（登録商標）言語は、プログラムローダの実行もロードされたプログラムを管理する機能も提供していない。また、Java（登録商標）言語は、メモリの制約された組込み機器内でアプリケーションをダウンロードおよび管理する手段も提供していない。

【0004】 いくつかの現存するJavaテクノロジーは、プログラムローダ・タイプの機能をサポートしている。Sun Microsystemsのサーブレット（Servlet）API（Application Program Interface）は、Java（登録商標）動作可能ウェブ（Web）サーバがサーバの機能を動的に拡張することを可能にする。サーブレットAPIは、主に、ウェブサーバにおいて一般的なCGI機能に取って代るものとして生成された。サーブレットのCGIタイプのプログラムに対して優れている点として、プラットフォーム非依存性、再利用可能性（オブジェクト指向技術によりJavaクラスを再利用する機能）、性能効率（同じサーブレット・インスタンスが多くの要求を処理することができるサーブレットの構成可能な起動モード（CGIスクリプトに対する呼出毎に新たなプロセスを生成する必要があることとは対照的である））、および管理効率（SunのJavaベースのウェブサーバは、新たなサーブレット・クラスの追加、およびサーブレットの開始並びに停止を容易に管理するJava adminアプレットを提供する）がある。

【0005】 サーブレットAPIは、すべてのサーブレットが3つのメソッド、すなわちinit()（初期化）、service()（サービス）およびdestroy()（破壊）を実行するよう指令する（mandate）ことにより、サーブレットのライフサイクルを定義する。init()メソッドは、サーブレットが最初に起動される時に呼出され、service()メソッドは、クライアントからの各要求を処理するために呼出され、destroy()メソッドは、サーブレットが停止している時（すなわち、ウェブサーバがシャットダウンしている時）に呼出される。サーブレットAPIは、メモリ管理を直接に管理しておらず、サーブレットを管理するための公式パッケージも提供していない。

【0006】 Hewlett Packardの組込みJava Lab SmallWebは、JVMで実行されるJavaベースのオブジェクトに対しウェブベースのインタフェースを提供する。SmallWebは、必要に応じてオブジェクトをロードすることができ、オブジェクトがその機能を（メソッド呼出により）ウェブブラウザにエクスポートする手段を提

供する。SmallWebは、明示的には、アプリケーションオブジェクトの停止またはメモリ管理機能を提供していない。更に、SmallWebには、追加ファイルシステムが必要であり、組み込み環境によっては、SmallWebのオーバーヘッド要求が大きすぎる場合がある。

【0007】多くの組み込みドメインにおいてJava（登録商標）を使用する問題の1つは、Java（登録商標）メモリ・サブシステムの非決定性の面である。Java（登録商標）言語の非決定性メモリ管理方式により、ガーベッジ・コレクタ・メソッドgc()による参照されないオブジェクトの回収（reclamation）が可能となるが、ガーベッジ・コレクタは、これらのオブジェクトがどのようにまたはいつ回収される（reclaimed）かを指定しない。ネイティブアプリケーションは、一般に、Java（登録商標）Runtime class gc()メソッドを呼出すことによりガーベッジ・コレクションを発生させる。JVMがメモリを使い尽くした場合、OutOfMemoryError（アウトオブメモリ・エラー）例外が発生する。メモリを管理する必要がないことがJava言語の利点ではあるが、大抵の組み込みアプリケーションでは、メモリがどのように管理されるかに対し、Java（登録商標）言語によって現在提供されている以上に厳しい制御が必要である。そのようにより厳しい制御が必要であるのは、ローメモリ（low memory）状態で実行を続けなければならない組み込みアプリケーションがあるためである。一般に、これは、継続して実行するために十分なメモリが空き状態であることを保証するよう、ネイティブの非Java組み込みアプリケーションにメモリマネージャを実行することによって処理される。

【0008】従って、Java動作可能な組み込み機器にダウンロードを行い、そのような組み込み機器におけるアプリケーションのライフタイムを制御する汎用的な方法が必要とされている。また、機器上でのアプリケーションの実行中に検出されるローメモリまたはノーメモリ（no-memory）状態を処理すると共に、プライオリティベースのアルゴリズムに従ってメモリを解放するメモリ管理ハンドラも必要とされている。

【0009】

【課題を解決するための手段】本発明は、メモリの制約された組み込み機器環境においてアプリケーションのダウンロードおよびライフサイクルを管理する新規なシステムおよび方法である。Javaベースのアプリケーションマネージャは、組み込み機器にインストールされた単一のJava仮想マシン（JVM）内で実行されるJavaアプリケーションのダウンロード、実行およびキャッシングを制御する。Java（登録商標）動作可能な組み込み機器におけるクラスファイルのロード、アプリケーションの実行の開始、初期化および停止、およびメモリ管理のための機能を提供するネットワーク対応のアプリケーション・プログラム・インタフェース（Applicatio

nProgram Interface（API））が明細化されている。本発明により、組み込み機器を、使用するハードウェアのタイプに関係なく容易に再プログラムおよび管理することができる。

【0010】本発明は、図面に關連して行われる以下の詳細な説明によってより深く理解されよう。なお、図面において、同じ要素には同じ参照番号を用いて示している。

【0011】

10 【発明の実施の形態】組み込み機器用の新規なアプリケーションマネージャについて、以下詳細に説明する。本発明は、Java（登録商標）環境のコンテキストで説明されているが、当業者には、本発明の原理が、ハードウェア非依存型言語で書かれたコードを処理するハードウェア非依存型プロセッサを有するすべてのシステムに拡張できるということが理解されよう。

20 【0012】ここで図1を参照すると、ネットワーク14を介して組み込み機器20と通信するコンピュータシステム12を備えたネットワークシステム10が示されている。ネットワーク14は、LANまたはWAN等、従来のランドリンクされた（land-linked）有線ネットワークであっても、無線ネットワーク、またはそれらの組合せであってもよい。

30 【0013】本システムにおけるコンピュータシステム12は、クライアントとして動作し、クライアントアプリケーション2を実行する。このクライアントアプリケーション2は、クライアントインタフェース4を介して、ネットワーク14を通じてサーバ機器（例えば、組み込み機器20）とインタフェースする。この例では、クライアントアプリケーション2は、組み込み機器20に対して要求（例えば、ダウンロード・アプリケーション3（download Appl3）、スタート・アプリケーション3（start Appl3）、ストップ・アプリケーション2（stop Appl2））を送信する。組み込み機器20は、その要求をサービスするサーバとして動作する。クライアントアプリケーション2は、コンピュータシステム12における独自の環境で実行してもよく、あるいは、それ自身のJava仮想マシン（JVM）（図示せず）を含むJava動作可能なウェブ（Web）ブラウザ（図示せず）内で実行してもよい。ウェブブラウザは、主なクライアントアプリケーション・アプレットのクラスファイルの場所を指定する組み込みJavaアプレットを用いて、ウェブ（Web）ドキュメントを解釈する。ウェブブラウザは、JVMを起動し、クライアントアプリケーション・アプレットのクラスファイルの場所を自身のクラスロードに渡す。各クラスファイルは、必要とする追加のクラスファイルの名前を知っている。これら追加のクラスファイルは、ネットワーク14（すなわち、ネットワーク14に接続された他のマシン）から入手されるようにしてもよく、あるいは、クライアントであるコンピュー

タシステム12から入手されるようにしてもよい。

【0014】組込み機器20は、Java仮想マシン(JVM)22がインストールされたJava動作可能機器である。図2は、組込み機器20をより詳細に示すブロック図である。組込み機器20は、JVM22、アプリケーションキャッシュ52およびデータキャッシュ54を有するメモリ50、およびネットワークインタフェース25を備えている。JVM22は、クラスローダ42および実行ユニット46を備えている。好ましくは、JVM22は、バイトコードベリファイヤ44も備えているが、メモリの制約された機器によってはこれが実行されない場合もある。本発明によれば、クラスローダ42は、本発明のアプリケーションマネージャ24を使用して実行される。アプリケーションマネージャ24は、他のJavaベースのプログラムのダウンロード、実行およびキャッシングを行うJavaプログラムである。JVM22は、組込み機器20上で実行を開始する時、アプリケーションマネージャ24の実行を開始する。

【0015】アプリケーションマネージャ24は、ネットワークベースのクラスローダとして動作するサーバ機器であり、ネットワークプロトコル48によりネットワーク14を介してアプリケーションクラスファイル40(以下、クラスファイル40とする)を受取り、そのクラスファイル40中に含まれるアプリケーションクラスを活性化することができる。当業者には周知であるように、クラスファイル40は、Java(登録商標)コンパイラによって生成されるハードウェアアーキテクチャ非依存型バイトコードを有するファイルである。このバイトコードは、JVM22等のJVMによってのみ実行が可能である。アプリケーションマネージャ24は、非一般的ネットワークベースのクラスローダ(ClassLoader)であり、要求毎にネットワーク14からクラスファイル40をロードする。一般的なクラスローダによって通常行われるようにJava(登録商標)loadClasses() (ロードクラス) メソッドによってクラスファイル40をロードする代りに、アプリケーションマネージャ24は、サーバソケット等のネットワークプロトコル48で、クラスファイル40をダウンロードするという要求を待つ。ダウンロード要求が受信されると、アプリケーションマネージャ24はネットワーク14からクラスファイル40を受信し、それを解析してクラス28a、28bにし、Java(登録商標)ClassLoader.defineClass()メソッドを呼出す。クラス28a、28bは、Java(登録商標)ClassLoader.resolveClass()メソッドを呼出すことによって解析され、オブジェクトのインスタンス化の用意ができていないことを確認する必要がある。

【0016】図3は、アプリケーション26a、26b、26cをどのようにして実行させるかを示してい

る。図示しているように、これは、次のようにして実行される。すなわち、アプリケーションのJavaクラスであるクラスオブジェクト28a、28bをインスタンス化し、クラスオブジェクト28a、28bのインスタンスであるアプリケーションオブジェクト30a、30b、30cを生成し、その後アプリケーション26a、26b、26cを実行するためにアプリケーションオブジェクト30a、30b、30cのメイン(main)メソッドを呼出す。同じJava(登録商標)クラスの複数のインスタンスを生成することにより、同じアプリケーションの複数のインスタンスを同時に実行することができる。例えば、図3に示すように、アプリケーションオブジェクト30a、30bは、同じクラスオブジェクト28aのインスタンスであるが、アプリケーションオブジェクト30cは、異なるクラスオブジェクト28bのインスタンスである。従って、アプリケーション26a、26bは、同時に実行することができる同じアプリケーションの異なるインスタンスであるが、アプリケーション26cは、完全に異なるアプリケーションである。

【0017】アプリケーションマネージャ24は、ダウンロード、開始、停止、問合せおよびメモリ管理機能を提供する。これらの機能をネットワーク14を介して提供するために、組込み機器20は、ネットワークインタフェース25を有している。このネットワークインタフェース25は、クライアントであるコンピュータシステム12が組込み機器20のアプリケーションマネージャ24に要求を送信することができるようにするネットワークプロトコル48を実行している。好ましい実施の形態において、ネットワークインタフェース25は、アプリケーションマネージャ24によって指定される汎用Java言語アプリケーションプログラムインタフェース(Application Program Interface (API))を用いて、アプリケーションマネージャ24に通信する。汎用APIにより、コンピュータシステム12等のリモート機器に、組込み機器20の所望の管理を指定する機能が与えられる。付録Aは、本発明の実施の形態において定義されるAPIを列挙している。

【0018】特に、アプリケーションマネージャ24は、新たなアプリケーションがネットワーク14を介して組込み機器20にダウンロードされるのを可能にする。好ましい実施の形態では、これは、APIのloadApplicationClass() (ロードアプリケーションクラス) メソッドによって実行される。loadApplicationClass() メソッドは、Javaのクラスファイル40のダウンロードおよび検証、クラスファイル40に含まれるクラスオブジェクト28a、28bの組込み機器20におけるJava(登録商標)環境へのロードを処理し、Javaインタプリタまたはジャストインタイム(Just in Time (JIT))コンパイラ(図示せず)を用いてアプリケーション

ンオブジェクト30a、30b、30cをインスタンス化することにより、クラスオブジェクト28a、28bの実行を準備する。また、アプリケーションマネージャ24は、組込み機器20上のJavaのアプリケーション26a、26b、26cの開始および停止を管理することにより、コンピュータシステム12または他のネットワーク化した機器において、ユーザが、組込み機器20にロードされたJavaのアプリケーション26a、26b、26cの実行を制御することができるようにする。好ましい実施の形態では、これは、汎用APIのstartApp() (スタートアプリケーション) メソッドおよびstopApp() (ストップアプリケーション) メソッドにより実行される。また、アプリケーションマネージャ24は、いずれのクラスオブジェクト28a、28bが現在ロードされているか、いずれのクラスがアプリケーションオブジェクト30a、30b、30cを有しているか、および各アプリケーションオブジェクト (インスタンス) はどのような実行状態 (例えば、「初期化」、「実行中」、「終了」) にあるか等のアプリケーション情報の問合せを可能にする。これは、好ましい実施の形態では、各々appClasses() (アプリケーションクラス) メソッド、applications() (アプリケーション) メソッドおよびappInstances() (アプリケーションインスタンス) メソッドによって実行される。ロードされたクラスオブジェクト28a、28bの各々の名前は、属性として、好ましくはアプリケーションマネージャ24オブジェクト内のハッシュテーブルとして、アプリケーションリスト23に格納される。また、アプリケーションマネージャ24は、空きメモリの容量および総メモリの容量等、Java環境からの情報の問合せを可能にする。好ましい実施の形態では、この情報の問合せは、freeMemory() (フリーメモリ) メソッドおよびtotalMemory() (トータルメモリ) メソッドを用いて行われる。また、アプリケーションマネージャ24は、ロードされたアプリケーションがアンロードされなければならない前に必要な空きメモリの容量に制限を設定する機能、およびキャッシングされたアプリケーションをアンロードすべき順序を設定する機能を含む。メモリ管理機能を提供する。これは、好ましい実施の形態では、setFreeMemoryLimit() (セットフリーメモリリミット) メソッドおよびsetFreeAppsFirst() (セットフリーアプリケーションファースト) メソッドを用いて実行される。空きメモリの制限およびアプリケーションをアンロードすべき順序については、getFreeMemoryLimit() (ゲットフリーメモリリミット) メソッドおよびgetFreeAppsFirstPolicy() (ゲットフリーアプリケーションファーストポリシー) メソッドを用いて問合わせることができる。この情報は、アプリケーションマネージャ24オブジェクト内のハッシュテーブルに、各アプリケーションリスト23エントリ毎に別々のフィールドとして、または別々のア

ンロードプライオリティリスト21として格納される。  
【0019】ネットワークプロトコル48を用いて、ネットワークインタフェース25を介するアプリケーションマネージャ24に対するネットワークアクセスが提供される。ネットワークプロトコル48は、いくつかの他の実施の形態のうちの任意のものによって実現するようにしてもよい。

【0020】第1の実施の形態では、ネットワークプロトコル48は、Java (登録商標) リモートメソッド呼出 (Remote Method Invocation (RMI)) を用いて実現される。Java RMIにより、他のJVMに常駐しているオブジェクトにおいて直接のメソッド呼出を行うことができる。一般に、RMIには、オブジェクトのシリアライズ、識別およびメソッドのディスパッチをサポートするために大容量のメモリが必要である。組込み機器が、RMIネームサーバの実行のサポートを含む実行中のRMIの高メモリアバヘッドをサポートする場合、RMIを選択して実現するということは、組込み機器20およびアプリケーションマネージャ24と対話

(interact) するクライアントアプリケーションがJavaベースである場合に適している。他の分散オブジェクトメカニズム (例えば、CORBA、DCOM) とRMIとの間には、ブリッジが存在し、それにより、クライアントにおいて他の言語で同様に書込みことが可能となる。アプリケーションマネージャ24は、RMIを使用してリモートクライアント12から直接の呼出を受入れることにより、アプリケーション26a、26b、26cをロードし、開始し、管理することができる。RMIベースのアプリケーションマネージャインタフェースの簡単な例を、付録Bに示す。

【0021】第2の他の実施の形態では、ネットワークプロトコル48はネットワークソケット (例えば、TCP/IPソケットプロトコル) を用いて実現される。クライアント12 (コンピュータシステム12) 側と同様にJava (登録商標) が使用される場合、ネットワークプロトコル48がクライアント12 (コンピュータシステム12) 側とサーバ (組込み機器20) 側との両方で一貫していることを保証するために、Javaクラス内にネットワークプロトコル48をラップする (wrap) ことは有益である。クライアント12 (コンピュータシステム12) またはアプリケーションマネージャ24が停止する可能性のあるネットワークの読出のブロック化を、ネットワークプロトコル48が実行しないことを保証するために、好ましくは、Java (登録商標) Socket.setSoTimeout() メソッドにより、またはJava (登録商標) InputStream.available() メソッドを用いてブロック化無しにどれくらいのデータが読出されるかを判断することにより、read() (読出し) 動作のための待ちのタイムアウトが設定されている。

【0022】かかるネットワークソケットプロトコル方

式の1つの実現例では、呼出されるメソッドを識別する単一バイトと、それに続く、ラッパ(wrapper)によりその特定のメソッドに対してデコードされる追加のデータとが、送信される。例えば、クラスファイル40をダウンロードするために、クライアント12(コンピュータシステム12)は、その要求がダウンロードのためのものであることを指定する「D」キャラクタバイトと、それに続く送信されているデータの全体長と、それに続く組込み機器20のネットワークアドレスとを送信する。他の実現例では、実際のデータはナル終了キャラクタのクラス名であり、そのクラスがメインメソッドを含むか否かを指定するバイトがその後に続き、更にクラスデータを含むバイトがその後に続く。プロトコルのデコードを実行するために、仮想デコードメソッドをサポートするオブジェクトモデルが生成される。また、クライアント12(コンピュータシステム12)にメソッド呼出の結果を返すことを、使用されている同様のプロトコルを用いてカプセル化してもよい。このように、要求から結果が非同期に返され、クライアント12(コンピュータシステム12)は実際に、結果に対しネットワークサーバとして動作することができる。クライアント12(コンピュータシステム12)が1つの組込みJVM22に通信する(talk)のみであり、結果を待ってアイドル状態である場合、クライアント(コンピュータシステム12)が、結果が同期して戻ってくるのを待つ(タイムアウト有りで)、より単純な方式を使用することができる。

【0023】好ましい実施の形態では、アプリケーションマネージャ24は、Javaネットワークサーバ-アプリケーションとして実現され、Java(登録商標)リモートメソッド呼出(RMI)プロトコル、ウェブがホストする要求のためのハイパーテキスト転送プロトコル(HTTP)、またはTCP/IPソケットによるアプリケーションレベルプロトコル等のネットワークプロトコル48にコード化される要求を受入れる。上述したように、ネットワークプロトコル48の要求の1つのタイプは、アプリケーションのJavaのクラスファイル40のダウンロードである。好ましい実施の形態では、クラスファイル40のダウンロードは、loadAppClass()メソッドまたはloadAndInit() (ロードおよび初期化)メソッドを呼出すことによって実行される。上述したように、クラスファイル40は、JVM22がJavaクラスオブジェクト(アプリケーションオブジェクト30a、30b、30c)を生成するために使用するバイナリのJavaバイトコードストリームである。アプリケーションクラスファイル40のメモリ50へのダウンロードに加えて、アプリケーション26a、26b、26cが継承するあらゆる基底クラス、またはそのクラスファイル40に含まれるアプリケーションクラス28a、28bとのインタフェースもまた、メモリ50の

アプリケーションキャッシュ52にダウンロードされる。アプリケーションマネージャ24がクラスファイル40のメモリ50へのキャッシングを可能にするため、ネットワークプロトコル48は、startApp() (スタートアプリケーション)メソッド呼出が後に続くinitApp() (初期化アプリケーション)メソッドにより、メモリ50に常駐している既にダウンロードされているクラスファイルの実行を可能にする。クラスファイル40がダウンロードされると、Java(登録商標)ClassLoader APIを用いて、クラスファイル40に関連するクラスオブジェクト28a、28bがインスタンス化される。ClassLoader.defineClass()メソッドは、クラスファイルのバイトアレイからクラスオブジェクト28a、28bを構成する。クラスは、定義されると、クラスのリンク化を実行するために解析されなければならない。それによってインスタンスオブジェクトが生成されメソッドが呼出される。解析プロセスは、Java(登録商標)ClassLoader.resolveClass() APIを呼出すことによって起動される。

【0024】クラスオブジェクト28a、28bがロードおよび定義されると、一例としての実施の形態ではJava(登録商標)Class.newInstance() APIにより、クラスオブジェクト28a、28bのインスタンスであるアプリケーションオブジェクト30a、30b、30cが生成される。アプリケーションオブジェクト30a、30b、30cは、アプリケーション26a、26b、26cを開始するために所望のメソッドを呼出す必要がある。newInstance() (ニューインスタンス)を呼出すために、クラスオブジェクト28a、28bは、属性をとらないデフォルトのコンストラクタを有していなければならない。

【0025】アプリケーション26a、26b、26cは、予め生成されたアプリケーションオブジェクト30a、30b、30cの所望のメソッドを呼出し、そのメソッドに属性を渡すことによって実行される。アプリケーションマネージャ24は、オブジェクトの呼出すメソッド、および渡す属性(あるとすれば)が何であるかを知らなければならない。これは、いくつかの方法のうちの1つによって実行される。

【0026】1つの実施の形態では、すべてのアプリケーションは、呼出すメソッドを指定するJavaインタフェースを実行する必要がある。例としてのインタフェースを以下に示す。

```
public interface AppBase
{
    public void main(InetAddress 0);
}
```

このインタフェースは、アプリケーション26a、26b、26cの実行時に呼出される最初のメソッドであるインスタンスメソッド「main()」を指定する。それがイ



ンスタンスメソッドであるため、クラスオブジェクト28a、28bのアプリケーションオブジェクト30a、30b、30cは、アプリケーションマネージャ24により、main()を実行する前に生成されなければならない。main()のみが呼出されているため、このオブジェクトはアプリケーションマネージャ24のJava（登録商標）ApplBaseオブジェクトとしてもよい。この例では、main()メソッドは、クラスをダウンロードしたクライアント（コンピュータシステム12）のネットワークアドレスを含むInetAddress引数を受取る。このように、アプリケーションは、必要ならば周知のポートによりクライアント（コンピュータシステム12）に戻る方向に通信することができる。

【0027】別の実施の形態では、JVM22が例えばJava（登録商標）RMIによりリフレクションをサポートする場合、メソッド名がアプリケーションマネージャ24にダウンロードされ、リフレクションを用いて所望のメソッドのアプリケーションクラスが探索される。クラスオブジェクト28a、28bがロードされると、所望のメソッドを見つけるためにそのクラスのJava（登録商標）getDeclaredMethod()が呼出される。getDeclaredMethod()はMethodオブジェクトを返すものであって、Methodオブジェクトは、invoke()（呼出）を呼出すことにより、そのメソッドを実行するために使用することができる。

【0028】更に他の実施の形態では、すべてのアプリケーションが継承する元となる共通の基底クラスから拡張されるメソッドが呼出される。

【0029】アプリケーションマネージャ24は、実行中となると、アプリケーション26a、26b、26cの新たな実行状態を記録する。各クラスオブジェクト28a、28bは、そのクラスの存在する各インスタンスを識別する各々のインスタンスリスト29a、29bを有している。インスタンスリスト29a、29bの各エントリは、現アプリケーション26a、26b、26cのライフサイクルの間にアプリケーションマネージャ24によって更新される実行状態変数を含む。

【0030】アプリケーションマネージャ24は、アプリケーション26a、26b、26cで発生するJava（登録商標）例外またはエラーのすべてを処理する。そうしなければ、JVM22は終了することとなる。アプリケーションマネージャ24は、アプリケーションが終了すると、そのアプリケーション26a、26b、26cがもはや実行していないということを（インスタンスリスト29a、29bにおいて識別されるインスタンスに対応する実行状態を更新することにより）記録し、クラスオブジェクト28a、28bがメモリ50内にキャッシングされるものでない場合、そのクラスオブジェクト28a、28bがガーベッジ・コレクションされるようにする。

【0031】クラスオブジェクト28a、28bのキャッシングは、性能をより優れたものとする本発明の独特の特徴である。アプリケーション26a、26b、26cが実行される毎にアプリケーションクラス（クラスオブジェクト）をダウンロードする代わりに、アプリケーションマネージャ24は、好ましくは、デフォルトでクラスオブジェクト28a、28bをキャッシングし、unloadApp1()（アンロードアプリケーション）メソッドを用いて明示的に要求される場合、またはメモリ管理ハンドラ27がローメモリまたはノーマemory状態の結果としてアンロードすべきクラスを選択する場合にのみ、クラスオブジェクト28a、28bをアンロードする。好ましい実施の形態では、メモリ管理ハンドラ27によって現在ロードされているクラスがアンロードされる順序は、setFreeAppsFirst()メソッドを用いて設定される。setFreeAppsFirst()メソッドにより、クライアントは、ローメモリまたはノーマemory状態の場合にアンロードされるアプリケーションの順序を設定または変更することができる。これは、クラスをアンロードするためのアプリケーションの順位付けを指示するアプリケーションリストにおいて、各エントリにアンロードプライオリティフィールドを付加する等、いくつかの方法のうちの任意の方法によって達成することができる。アプリケーションマネージャ24は、クラスをキャッシングすることにより、クラスに対する参照（reference）を保持し、それによって、Java仮想マシン22に、クラスオブジェクト28a、28bに対するガーベッジ・コレクションを行わせない。そして、クラスオブジェクト28a、28bを再利用して、クラスオブジェクト28a、28bの新たなインスタンス30a、30b、30cを生成し、アプリケーション機能を再実行することができる。

【0032】また、アプリケーションマネージャ24は、好ましくは、アプリケーションが終了および/またはアンロードされた後であっても、メモリ50内のデータキャッシュ54へのアプリケーションデータのキャッシングを可能にする。例として、電子テストメインにおけるJavaアプリケーションがある。アプリケーションが最初に実行されるとセットアップ情報を保存することができ、それによって、そのアプリケーションを将来実行する場合により高速に実行することができる。この例では、アプリケーションマネージャ24は、テストシステム較正（calibration）情報をキャッシングすることができる。更に、テストシステム較正情報をデータキャッシュ54にキャッシングすることにより、アプリケーションマネージャ54はデータへの参照を保持し、それによってJava仮想マシンにデータをガーベッジ・コレクトさせないようにする。従って、この方法でのデータキャッシングにより、同じかまたは異なるアプリケーションの後続する実行のために情報を保存することができる。

【0033】アプリケーションマネージャ24は、アプリケーション26a、26b、26cをローメモリまたはノーマemory状態で実行を続けるよう試みる。JVM22が、アプリケーション26a、26b、26cの実行中にメモリを使い尽くした場合、OutOfMemoryErrorエラーが生成される。アプリケーションマネージャ24は、ローメモリまたはノーマemory状態を処理するメモリ管理ハンドラ27を有している。好ましい実施の形態では、メモリ管理ハンドラ27は、JVM22によって生成されるOutOfMemoryErrorの発生により実行するようトリガされる。1つの実施の形態では、メモリ管理ハンドラ27は、そのアプリケーションキャッシュからクラスオブジェクト28a、28bおよびアプリケーションオブジェクト30a、30b、30cを適切にダンプすることにより作用する。キャッシュされたクラスオブジェクト28a、28bおよびアプリケーションオブジェクト30a、30b、30cをアンロードする順序は、setFreeAppsFirst()メソッドを用いて設定される。あるいは、その順序は、メモリ管理ハンドラ27自身においてコード化された予め決められたアルゴリズムに従って決定される。アプリケーションマネージャ24は、Java（登録商標）Runtime.gc()メソッドを呼出すことにより、アンロードされたクラスオブジェクト28a、28bおよびアプリケーションオブジェクト30a、30b、30cが回収（reclaim）可能であることをJVM22に通知する。アプリケーションキャッシュ52からクラスオブジェクト28a、28bをアンロードすることにより、そのクラスオブジェクト28a、28bに関連するアプリケーションオブジェクト30a、30b、30cが同様に終了およびアンロードされる。従って、アプリケーション実行の起動速度が重要である場合、クラスオブジェクト28a、28bをアンロードする前にアプリケーションオブジェクト30a、30b、30cをアンロードすることが好ましい。一方で、特定のアプリケーションを無関係に実行させておくことが重要である場合、その特定のアプリケーションを実行させておくために十分なメモリを解放するべく、それらオブジェクトに関連するクラスと共に別のクラスのすべてのアプリケーションオブジェクトをアンロードすることが好ましい。アプリケーションを終了させることについての解決法は、ローメモリ状態に陥って別々にメモリレベルを監視するか、またはアプリケーションの結果をチェックポイントすることによりアプリケーションがチェックポイントの後に継続を再開することができるようにすることである。これらの方法により、実行中のアプリケーションは割込みされずに、または少なくとも可能な限り割込みされずに継続することができる。

【0034】アプリケーションマネージャ24は、好ましくは、メモリ管理機能を提供する。1つの実施の形態では、アプリケーションマネージャ24は、アプリケー

ションキャッシュ52およびデータキャッシュ54各々におけるクラスオブジェクト28a、28b、アプリケーションオブジェクト30a、30b、30cおよびグローバルデータを参照する。それによって、それらの関連するオブジェクトがガーベッジ・コレクトされないことが保証される。

【0035】他の実施の形態では、メモリ管理ハンドラ27は、連続的に空きメモリレベルを継続して監視し、ローフリー（low-free）メモリまたはノーフリー（no-free）メモリ時には、そのキャッシュ52、54からオブジェクトを除去し、必要に応じてガーベッジ・コレクションをトリガして更にメモリを解放する。上述したように、ガーベッジ・コレクションは、アプリケーションキャッシュ52またはデータキャッシュ54からアイテムを除去した後にJava（登録商標）Runtime.gc()メソッドを呼出すことによりトリガされる。アプリケーションマネージャ24を実行する場合、それがローメモリ状態においてメモリ50をどのように管理するかについて、選択を行わなければならない。1つの選択は、ローフリーメモリが発生した時に、アプリケーションキャッシュ52またはデータキャッシュ54のいずれのキャッシュから最初にアイテムが除去すべきかということである。大抵のケースでは、アプリケーションオブジェクト30a、30b、30cおよび関連するクラスオブジェクト28a、28bを除去し、必要に応じてそれらを再ダウンロードの方が容易である。この判断は、グローバルデータを再生成するためにかかる長い時間に基づいている。

【0036】アプリケーションおよびグローバルデータオブジェクトは、好ましくは、キャッシング方式が正確に作用するために可能な限り別々であるように設計されている。グローバルデータクラスがクラスオブジェクト28a、28bのメンバとして参照される場合、クラスオブジェクト28a、28bもまた参照されないということがない限り、データクラスが参照されないということでもメモリは解放されない。グローバルデータクラスがクラスオブジェクト28a、28bを参照するという逆の場合によっても、同じ結果となる。クラスオブジェクト28a、28bは、クラスメンバではなくメソッドにおいてのみグローバルデータクラスを参照すべきである。好ましくは、グローバルデータオブジェクトは、クラスオブジェクト28a、28bを参照することはない。

【0037】図4は、アプリケーションキャッシュ52の管理の一例としての実施の形態を示す動作フローチャートである。アプリケーションマネージャ24は、クライアントであるコンピュータシステム12から要求を受信する（402）。要求の実行により空きメモリを使用すると（例えば、loadApp1()、loadAndInit()、InitApp1()）、受信した要求の実行によりローメモリまたはノ

10

20

30

40

50

ーメモリ状態となったか否かを判断する(404)。その場合、アプリケーションキャッシュ52からアンロードするクラスオブジェクト28a、28bおよび/またはアプリケーションオブジェクト30a、30b、30cを選択する(406)。そして、選択したオブジェクトをアプリケーションキャッシュ52からアンロードする(408)。受信した要求がloadApp()またはloadAndInit()要求である場合、ネットワーク14からクラスオブジェクトをロードし定義する(410)。要求がloadApp()要求である場合、その要求は完了する。要求がloadAndInit()要求またはInitApp()要求である場合、アプリケーションオブジェクト30a、30b、30cをインスタンス化し(412)、その要求は完了する。要求がstartApp()要求である場合、概してアプリケーションオブジェクト30a、30b、30cのmain()メソッドを呼出すことにより、アプリケーションを開始する(414)。要求がstopApp()要求である場合、アプリケーションを停止する(416)。要求がunload()要求である場合、クラスオブジェクト28a、28bをアンロードする(418)。要求がsetFreeMemoryLimit()要求である場合、ローメモリ状態の判断の基準となる空きメモリ閾値を設定する(420)。要求がsetFreeAppsFirst()要求である場合、ローメモリまたはノーマル状態が検出された時にアンロードのためにアプリケーションクラス28a、28bおよびオブジェクトクラス30a、30b、30cが選択される順序を設定する(422)。要求が問合せ(例えば、いずれのアプリケーションクラスがロードされるか(applClasses())、いずれのアプリケーションが初期化されるか(application s())、いずれのアプリケーションが現在実行中であるか(applInstances())、どのくらい使用可能なメモリが存在するか(freeMemory())、どのくらい総メモリが存在するか(totalMemory())、現在の空きメモリ制限閾値は何であるか(getFreeMemoryLimit())、またはクラスおよびアプリケーションオブジェクトをアンロードするための現在の順序は何か(getFreeAppsFirstPolicy()))、その問合せを実行し要求されたパラメータを返す(424)。

【0038】ローメモリまたはノーマル状態が存在すると決定した場合(404)、一例としての実施の形態では、アプリケーションマネージャ24は、アンロードするよう選択されているオブジェクトに対する参照を除去し、Java(登録商標)Runtime.gc()メソッドを呼出すことにより、参照されないオブジェクトがアプリケーションキャッシュ52から除去されるようにする。上述したように、Java(登録商標)Runtime.gc()メソッドによって参照されないクラスのみをガーベッジ・コレクトすることができるため、参照されないクラス28a、28bのみがアプリケーションキャッシュ52から除去される。JVM22の実現例によっては、そのクラ

スのClassLoaderオブジェクトもまたガーベッジ・コレクトされるクラスとして参照されてはならないものとしてもよい。

【0039】図5は、本発明を採用するシステム500の一例として実現例を示すクラス図である。図5の一例としての実現例において定義および使用されるクラスの各々の記述を、以下の表1に示す。表2には、図5の一例としての実現例において使用されるクラスの各々において定義される主な属性の記述が含まれている。図5に示すように、クラスAppMar502は、クラスApplication510の1対多のインスタンスを管理する。AppMar502は、クライアントクラスAppClient508とインタフェースするために、クラスClientProtocol506に通信するクラスServerProtocol504を使用する。表1に記述しているように、AppMar502は、アプリケーションマネージャ24を実現するクラスである。Application510は、単一のアプリケーションをモデル化している。ServerProtocol504およびClientProtocol506は、クライアントインタフェース4を実現する。それらは共に、アプリケーションマネージャクラスAppMar502とクライアントクラスAppClient508との間のネットワークプロトコル48をカプセル化する。

【0040】表1

クラス: AppMar

記述: アプリケーションマネージャ全体のクラス

クラス: Application

記述: 単一のアプリケーションをモデル化

クラス: ServerProtocol, ClientProtocol

記述: AppMarによってサポートされるインタフェースを表すAppMarクラスとAppClientクラスとの間のネットワークプロトコルをカプセル化

クラス: AppBase

記述: AppMarによって使用されるためにApplicationが定義しなければならないメソッドを定義するインタフェース

クラス: AppClient

記述: AppMarクラスと対話する一例としてのクライアント側アプリケーション

【0041】表2

名前: AppMar.idata

カテゴリ: 属性

記述: Applicationクラスのライフタイムを超えて存続する必要のあるグローバルデータ値のハッシュテーブル。getメソッドおよびsetメソッドによりアクセスされる。

名前: AppMar.iappls

カテゴリ: 属性

記述: AppMarが管理するよう指示されたApplicationオブジェクトの集まり

名前: Application.rclasses

カテゴリ：属性

記述：このApplicationに関連するクラスのハッシュテーブル。クラスは名前によって一意に識別される。クラスの階層構造を用いてアプリケーションを生成することができるが、それにはハッシュテーブルの使用が必要である。

名前：Application.iapps

カテゴリ：属性

記述：現在実行中のアプリケーションのインスタンスの集まり。インスタンスは、クラス名によって識別され、インスタンスカウンタは生成される各インスタンスについてインクリメントされる。

名前：Application.iMainClass

カテゴリ：属性

記述：実行されるmain()ルーチンを含むクラス。実行されるアプリケーションオブジェクトは、このクラスから生成される。

名前：ServerProtocol.iSock

カテゴリ：属性

記述：AppMqrとの通信のために使用されるServerSocket 20

名前：ServerProtocol.iAppMqr

カテゴリ：属性

記述：ServerProtocolはあらゆる要求をデコードし、その要求を処理するためにAppMqrに対し適当な呼出を行う。あるいは、ServerProtocolに代えて、AppMqrに対する直接のRM[]呼出でもよい。

名前：ClientProtocol.iSock

カテゴリ：属性

記述：ClientProtocolがServerProtocolと通信する手段 30

【0042】図6は、新たなクラスのロードを実行するためのクラス間の通信を示すシステム500のブロック図である。表3は、その初期の仮説、ロード動作の結果、およびロードを実行するために必要なエージェントを指定している。ここに示すように、アプリケーションマネージャクラスであるAppMqr502は、組込み機器20のJVM22において実行中であり、かつ、ユーザが、クライアントであるコンピュータシステム12で実行中のAppClient508に対し新たなクラスを組込み機器20にダウンロードするよう命令することにより、ダウンロードを要求したものと仮定する。

【0043】表3

仮説：AppMqrは既にJVMで実行中である。

結果：アプリケーションに関連するクラスは、JVMにロードされており、「メインクラス」のインスタンスが生成され、この新たなインスタンスにおいてinit()メソッドが呼出される。

エージェント：

User=AppClientに対し新たなアプリケーションクラス 50

をAppMqrを実行している組込み機器にダウンロードするよう命令

AppClient=アプリケーションクラスをダウンロードするホスト側プログラム

ClientProtocol=クライアントのためのネットワーク通信を処理するホスト側クラス

ServerProtocol=AppMqrのためのネットワーク通信を処理するサーバ側クラス

AppMqr=アプリケーションマネージャクラス

Application=ダウンロードされたアプリケーションをモデル化するアプリケーションクラス

図6に示すように、ダウンロード動作は、クライアントユーザクラスAppClientU516等のクライアントユーザアプリケーションからクライアントアプリケーションAppClient508へのユーザ要求602によって開始する。この例では、クライアントユーザクラスであるAppClientU516は、「D」キャラクタと、それに続くクラスデータを含むファイルの名前と、更にそれに続くクラスデータを送信すべきネットワークノードとを送信するプロトコルに従うことにより、ダウンロード要求を生成する。この例では、クラスデータは「test」と名づけられ、組込み機器20のネットワークノードアドレスは、「HP」と名づけられている。ユーザ要求602により、AppClient508が実行され、クラス「test」がノード「HP」にダウンロードされる。AppClient508は、「test」に含まれるクラスデータをメモリに読出し、その後ClientProtocol.loadClass()メソッドを呼出してクラスをダウンロードする(604)。ClientProtocol506は、ノード「HP」上のServerProtocolへのネットワーク14コネクションを形成し、ネットワーク14を介してクラスデータファイル「test」に関連するクラスデータを送信する(606)。ServerProtocol504は、要求およびクラスデータをデコードし、その後AppMqr.loadClass()メソッドを呼出してクラスをロードする(608)。AppMqr502は、必要ならばクラスに対してApplicationオブジェクトを生成する(610)(すなわち、クラスは既にアプリケーションキャッシュにキャッシングされていない)。そして、AppMqr502は、Application510のloadClass()メソッドを呼出し(612)、それにクラス「test」に関連するすべてのクラスデータを渡す。新たなクラスオブジェクトが、ApplicationsのAppMqrキャッシュに付加され(614)、それが管理するそのアプリケーションオブジェクトへのクラス「test」の付加を反映するようにアプリケーションリストAppMqr.iapps23が更新される(614)。ロードされているクラス「test」がメインクラスである場合、Application510はクラスを定義して解析し、クラスのインスタンスを生成し、インスタンスのinit()メソッドを呼出す(618)。新たなアプリケーションオブジェクトが、クラスキャッシュに付加され

(616)、その新たなアプリケーションオブジェクトの付加を反映するようにインスタンスリスト29が更新される(616)。真の結果は、クラスのロードが成功したことを通知してAppClient508に戻される(620)。

【0044】実施例において、必要ならば、ServerProtocol504は、各要求を処理するための別々のスレッドを生成し、複数の要求が同時に処理されるようにする。このようにして実現された場合、AppMar502およびApplication510によって実現される方法は、適切に同

期しなければならない。  
【0045】Application510が複数のクラスから構成されている場合、各クラスに対しloadClass()要求が作成されることとなる。これには、AppMar502が既存のApplication510にクラスを配置および加えることができる必要がある。アプリケーションは、メインクラスの名前と同一の一意の名前によって識別される。慣習的に、基底クラスおよびインタフェースは、メインクラスをロードする前にロードされる。これにより、メインクラスのインスタンスが正確に生成されることが保証される。別の方法は、アプリケーションのインスタンスを生成するために新たなAppMarメソッドを生成するとい

うものである。これにより、同じクラスの複数のアプリケーションが同時に実行されることが可能になる。  
【0046】図7は、アプリケーションの開始を実行するためのクラス間の通信を示すシステム500のブロック図である。表4は、初期の仮説、開始動作の結果、および開始を実行するために必要なエージェントを指定している。ここに示すように、アプリケーションマネージャクラスであるAppMar502は、組込み機器20のJVM22において実行中であり、かつ、ユーザが、そのユーザのマシンで実行中のAppClient508に対し組込み機器20でアプリケーションを開始するよう命令することにより開始を要求したものと仮定する。

【0047】表4

仮説：AppMarは既にJVMで実行中である。AppMarにより、アプリケーションクラスが予めロードされている。

結果：所望のApplicationが実行を開始する。

エージェント：

User=AppClientに対し、ノード「HP」でアプリケーション「test」を開始するよう命令。

AppClient=Applicationsを開始するホスト側プログラム

ClientProtocol=クライアントのためのネットワーク通信を処理するホスト側クラス

ServerProtocol=AppMarネットワーク通信を処理するサーバ側クラス

AppMar=アプリケーションマネージャクラス

Application=開始すべきアプリケーションをモデル化

するアプリケーションクラス

AppBase=開始中のアプリケーション・インスタンス  
図7に示すように、開始動作は、クライアントユーザクラスAppClientU516等のクライアントユーザアプリケーションからクライアントアプリケーションAppClient508へのユーザ要求702によって開始する。この例では、クライアントユーザクラスAppClientU516は、「B」キャラクタ(「begin(開始)」に対応)

と、それに続くアプリケーション名「test」と、更にそれに続くクラスデータが送信すべきネットワークノード「HP」とを送信するプロトコルに従うことにより、開始要求を生成する。要求702により、AppClient508が実行され、ノード「HP」でアプリケーション「test」が開始される。AppClient508は、ClientProtocol.startApp()メソッドを呼出してアプリケーションを開始する(704)。ClientProtocol506は、ノード「HP」におけるServerProtocol504へのネットワークコネクションを形成し、開始メッセージを送信する(706)。ServerProtocol504は、開始要求をデコードし、その後AppMar.startApp()メソッドを呼出す(708)。AppMar502は、適切なApplication510を配置し、その後Application510にstartApp()メソッドを呼出す(710)。Application510は、新たなThread(スレッド)518を生成(712)してそれ自身のrun() (実行)メソッドを実行する(714)。run()メソッド内では、AppBase514のメインメソッドが呼出され(716)、種々のパラメータが渡される。Application510にいかなるパラメータも渡されない場合、run()メソッドを直接に呼出すことができる。そして、Application510は、アプリケーションインスタンスの状態を「実行中」に変更する(718)。

【0048】他の実施の形態では、AppClassオブジェクト(すなわち、ClassLoader512)が、Application510のインスタンスをモデル化するApplicationクラスを管理する。

【0049】図8は、アプリケーションの停止を実行するためのクラス間の通信を示すシステム500のブロック図である。表5は、初期の仮説、停止動作の結果、および停止を実行するために必要なエージェントを指定している。ここに示すように、アプリケーションマネージャクラスAppMarは組込み機器20のJVM22において実行中であり、かつ、ユーザが、そのユーザのマシンで実行中のAppClientに対し組込み機器20でアプリケーションを停止するよう命令することにより停止を要求したものと仮定する。

【0050】表5

仮説：AppMarは、既にJVMにおいて実行中である。AppMarによりアプリケーションクラスが予めロードされている。アプリケーションは既に実行中である。

結果：所望のApplicationが実行を停止する。

エージェント：

User=AppClientに対しノードHPでのアプリケーションtestの停止を命令

AppClient=Applicationsを開始するホスト側プログラム

ClientProtocol=クライアントに対するネットワーク通信を処理するホスト側クラス

ServerProtocol=AppMgrネットワーク通信を処理するサーバ側クラス

AppMgr=アプリケーションマネージャクラス

Application=停止すべきAppBaseインスタンスを含むアプリケーションクラス

AppBase=停止すべき実行中のアプリケーションインスタンス

図8に示すように、停止動作は、クライアントユーザクラスAppClientU516等のクライアントユーザアプリケーションからクライアントアプリケーションAppClient508へのユーザ要求802で開始する。この例では、クライアントユーザクラスAppClientU516は、「E」キャラクタ（「end（終了）」に対応）と、それに続くアプリケーション名「test」と、更にそれに続くクラスデータを送信すべきネットワークノード「HP」とを送信するプロトコルに従って、停止要求を生成する。ユーザ要求802により、AppClient508が実行され、ノード「HP」でのアプリケーション「test」の実行が停止する。AppClient508は、ClientProtocol.stopApp()メソッドを呼出してアプリケーションを停止する（804）。ClientProtocol506は、ノード「HP」でのServerProtocol504に対するネットワークコネクションを形成し、停止メッセージを送信する（806）。ServerProtocol504は、停止要求をデコードし、その後AppMgr.stopApp()メソッドを呼出す（808）。AppMgr502は、適切なApplication510を配置し、その後Application510にstopApp()メソッドを呼出す（810）。Application510は、stopApp()に渡された名前によってAppBase514インスタンスを配置し、そのterminate()（終了）メソッドを呼出す（812）。そして、アプリケーションは、Thread.join()を用いてAppBaseが終了したことを確認にする。Application510は、AppBase514インスタンスの状態を「終了」に変更する（814）。

【0051】アプリケーションはそれ自身のスレッドによって実行中であるため、AppBase514インスタンスの停止は、非同期イベントである。アプリケーションの停止を、実際には待たないことが望ましい。それは、アプリケーションが、その終了フラグをポーリングしない場合、停止しないためである。従って、実施例では、その代わりに、タイムアウトを用いてjoin()（ジョイン）メソッドを使用し、stop()を呼出すことにより強制的に終

了してもよい。

【0052】図9は、Application510のインスタンスの実行中にローメモリまたはノーマル状態が発生した場合のクラス間の通信を示すシステム500のブロック図である。表6は、初期の仮説、アウトオブメモリ状態の結果、およびその状態の処理を実行するために必要なエージェントを指定している。ここに示すように、アプリケーションマネージャクラスAppMgr502は組み込み機器20のJVM22において実行中であり、1つまたは複数のアプリケーションクラスがロードされ実行中であると仮定する。

【0053】表6

仮説：AppMgrは、既にJVMにおいて実行中である。1つまたは複数のアプリケーションクラスがロードされ実行中である。

結果：JVMおよびAppMgrは、実行を継続し、ユーザアプリケーションクラスおよびインスタンスキャッシュを管理することによりメモリを解放する。

エージェント：

20 User=AppClientに対し、ノード「HP」でのアプリケーション「stop」を停止するよう命令

AppClient=Applicationsを開始するホスト側プログラム

ClientProtocol=クライアントに対するネットワーク通信を処理するホスト側クラス

ServerProtocol=AppMgrネットワーク通信を処理するサーバ側クラス

AppMgr=アプリケーションマネージャクラス

Application=停止すべきAppBaseインスタンスを含むアプリケーションクラス

30 AppBase=停止すべき実行中のアプリケーションインスタンス

【0054】図9に示すように、実行中のApplication510のインスタンスAppBase514は、OutOfMemoryErrorを生成する（902）。好ましくは、OutOfMemoryErrorは、Application.run()メソッド内のメモリ管理ハンドラ27によって処理される（904）。メモリ管理ハンドラ27は、AppMgr.outOfMemory()メソッドを呼出す（906）ことにより、AppMgr502に対しその状態を通知する。そして、Application510は、run()から戻り、AppBase514スレッドを停止する（908）。AppMgr502は、そのアプリケーションキャッシュ52を通して、Application.appCount()メソッドを呼出す（908）ことにより実行中のAppBasesを有していないクラスを探す。AppBasesを実行していないそれらアプリケーションは、freeMemory()と命令され（910）、それにより、アプリケーションキャッシュ52からそれらのクラスをダンプし、すべてのオブジェクト参照をnullに設定する。その後、AppMgr502は、そのアプリケーションキャッシュ52からそれらア

アプリケーションを除去し(912)、Runtime.gc()を呼出してガーベッジ・コレクションによって参照されないオブジェクトが収集されるようにする(914)。これにより、メモリが解放される。

【0055】以上、本発明の実施例について詳述したが、以下、本発明の各実施態様の例を示す。

【0056】(実施態様1) 組込み機器(20)において1つ以上のアプリケーション(26a、26b、26c)を管理するアプリケーションマネージャ(24)であって、前記組込み機器(20)が、クラスオブジェクト(28a、28b)を格納するアプリケーションキャッシュ(52)と、ネットワーク(14)を介してクライアント(12)と通信するネットワークインタフェース(25)と、該組込み機器(20)にインストールされ該組込み機器(20)上で実行されるJava仮想マシン(JVM)(22)と、を備えるアプリケーションマネージャ(24)において、(a) エントリを有し、前記エントリの各々が、前記組込み機器(20)に現在ロードされているクラスオブジェクト(28a、28b)を識別するクラスオブジェクトリスト(23)と、(b) 前記ネットワークインタフェース(25)を介して前記クライアント(12)から受信するアプリケーションクラス(28a、28b)をロードし、前記アプリケーションクラスに対する新たなクラスオブジェクト(28a、28b)を生成し、前記アプリケーションキャッシュ(52)に前記新たなクラスオブジェクト(28a、28b)を格納し、前記新たなクラスオブジェクト(28a、28b)を前記アプリケーションキャッシュ(52)にロードされているものとして識別するために、前記クラスオブジェクトリスト(23)に新たなエントリを付加するクラスローダメソッド(loadApp(), loadAndInit())と、を具備するアプリケーションマネージャ(24)。

【0057】(実施態様2) 各クラスオブジェクト(28a、28b)は、前記クラスオブジェクトリスト(23)で識別されるものであり、エントリを有し、前記エントリの各々が、前記組込み機器(20)上で現在インスタンス生成されている前記クラスオブジェクトのインスタンス(30a、30b、30c)を識別するインスタンスリスト(29a、29b)を具備する、実施態様1記載のアプリケーションマネージャ(24)。

【0058】(実施態様3) 前記インスタンスリスト(29a、29b)の各エントリは、該エントリに対応する前記インスタンス(30a、30b、30c)の現在実行状態を示す実行状態属性を有する。実施態様2記載のアプリケーションマネージャ(24)。

\* 【0059】(実施態様4) 開始すべきインスタンス(30a、30b、30c)が存在しない場合、クラスオブジェクト(28a、28b)の新たなインスタンス(30a、30b、30c)を生成して、前記新たなインスタンス(30a、30b、30c)を前記クラスオブジェクト(28a、28b)の前記インスタンスリスト(29a、29b)に付加し、前記開始すべきインスタンスが実行を開始するようにし、前記開始すべきインスタンス(30a、30b、30c)が現在実行中であることを示すために、前記開始すべきインスタンスの前記インスタンスリスト(29a、29b)の前記エントリに対応する前記実行状態属性を更新する、開始アプリケーションメソッド(Init()、Start())を具備する。実施態様3記載のアプリケーションマネージャ(24)。

【0060】(実施態様5) 前記組込み機器(20)上で現在実行中であり、かつ、停止すべきであるインスタンス(30a、30b、30c)が実行を停止するようにし、前記停止すべきインスタンス(30a、30b、30c)が実行中でないことを示すために、前記停止すべきインスタンス(30a、30b、30c)の前記インスタンスリスト(29a、29b)の前記エントリに対応する前記実行状態属性を更新する停止アプリケーションメソッド(stopApp())を具備する、実施態様4記載のアプリケーションマネージャ(24)。

【0061】(実施態様6) ローメモリまたはアウトオブメモリ状態(OutOfMemoryError())の検出に対応して、前記アプリケーションキャッシュ(52)にキャッシングされたクラスオブジェクト(28a、28b)のうちアンロードすべきクラスオブジェクト(28a、28b)を選択し、前記アプリケーションキャッシュ(52)から前記選択したクラスオブジェクト(28a、28b)をアンロードするメモリ管理ハンドラ(27)を具備する、実施態様1から実施態様5記載のアプリケーションマネージャ(24)。

【0062】上に詳述したように、本発明は、アプリケーションマネージャおよびAPI仕様を提供し、電子機器または装置において、およびリソース制約が制限された環境を有する他の組込みシステムにおいて使用することが企図されたJava仮想マシン内で実行される。本発明は、より優先度の高いアプリケーションのためにメモリを解放すべくメモリリソースが制約されることとなった場合に、将来の実行のためにアプリケーションをキャッシングおよび終了する柔軟性を提供する。

【0063】以下に、付録A及び付録Bを添付する。

#### 付録A

JavaアプリケーションマネージャAPI

```
import java.io.IOException;
public interface AppMngr
```

```

{
// アプリケーション中心のメソッド
public String[] applClasses() throws IOException;
public String[] applications() throws IOException;
public String[] applInstances() throws IOException;
public void loadAppClass( String appName, String className,
                        boolean mainClass, byte classData[ ] )
                        throws IOException;
public String initApp( String appName ) throws IOException;
// 動作するメインクラスでなければならない
public String loadAndInit( String appName, String className,
                        byte classData[ ] ) throws IOException;
public String startApp( String appName, String appId ) throws IOException;
public void stopApp( String appName, String appId ) throws IOException;
public void unloadApp( String appName ) throws IOException;
// メモリAPI
public long freeMemory() throws IOException;
public long totalMemory() throws IOException;
public double getFreeMemoryLimit() throws IOException;
public void setFreeMemoryLimit( double percent ) throws IOException;
public boolean getFreeAppsFirstPolicy() throws IOException;
public void setFreeAppsFirst( boolean val ) throws IOException;
} // interface AppMgr 終了

```

【0064】

## 付録B

RMIベースのアプリケーションマネージャインタフェースの簡単な例

```

public interface AppMgr extends java.rmi.Remote

```

```

{
// アプリケーション管理 API
public void loadClass( String className, byte classData[ ], boolean main
) throws RemoteException;
public Object startApp( String className, String args[ ] ) throws RemoteException;
public void stopApp( Object app ) throws RemoteException;
unloadClass( String className );
//ロードされたクラスのリスト及び実行中のアプリケーション名の入手
public String[] enumerateAppClasses( ) throws RemoteException;
public String[] enumerateApps( ) throws RemoteException;
// JVM情報の入手 - システム及びランタイムクラスを参照
public long freeMemory( ) throws RemoteException;
public long totalMemory( ) throws RemoteException;
public Properties getProperties( ) throws RemoteException;
}

```

【図面の簡単な説明】

【図1】本発明が動作するネットワークシステムのシステム図である。

【図2】本発明が実行される組込み機器のブロック図である。

【図3】本発明に従って、アプリケーションが実行される過程を示す系統図である。

【図4】本発明に従って実現されるアプリケーションマネージャの1つの実施形態の動作フローチャートである。



【図5】本発明に従って実現されるアプリケーションマネージャの1つの実現例のクラスダイアグラムである。

【図6】ロード動作を実行するための図5に示すクラス間の通信を示す系統図である。

【図7】開始動作を実行するための図5に示すクラス間の通信を示す系統図である。

【図8】停止動作を実行するための図5に示すクラス間の通信を示す系統図である。

【図9】アウトオブメモリ状態を処理するための図5に示すクラス間の通信を示す系統図である。

【符号の説明】

12：コンピュータシステム

\* 14：ネットワーク

20：組込み機器

23：アプリケーションリスト

24：アプリケーションマネージャ

25：ネットワークインタフェース

26a、26b、26c：アプリケーション

27：メモリ管理ハンドラ

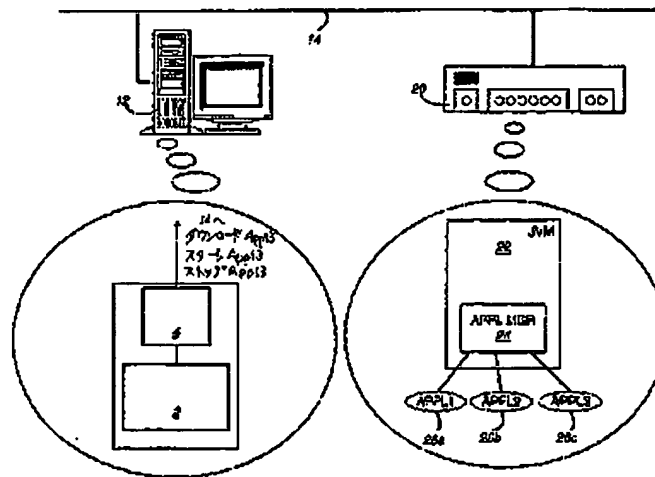
28a、28b：クラスオブジェクト

29a、29b：インスタンスリスト

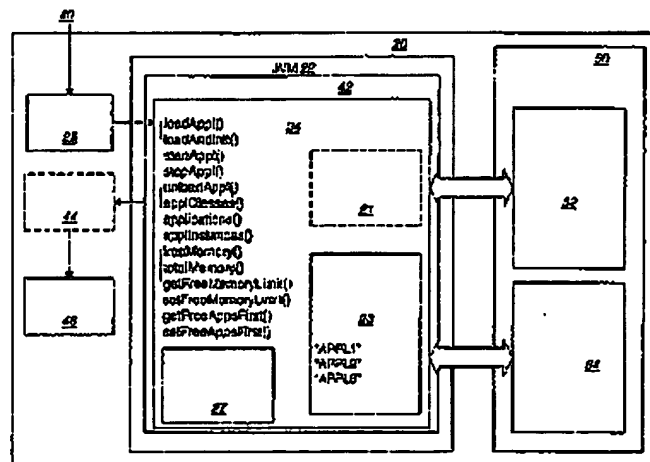
10 30a、30b、30c：アプリケーションオブジェクト

\* 52：アプリケーションキャッシュ

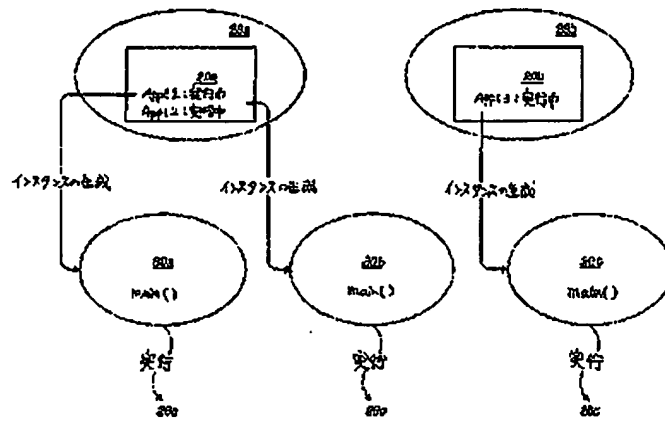
【図1】



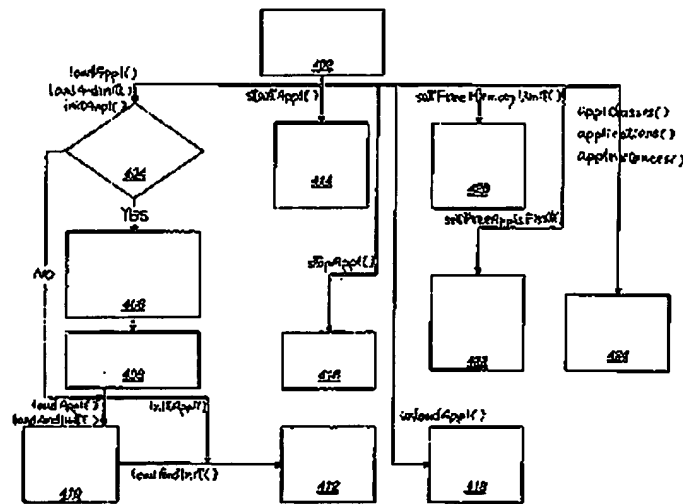
【図2】



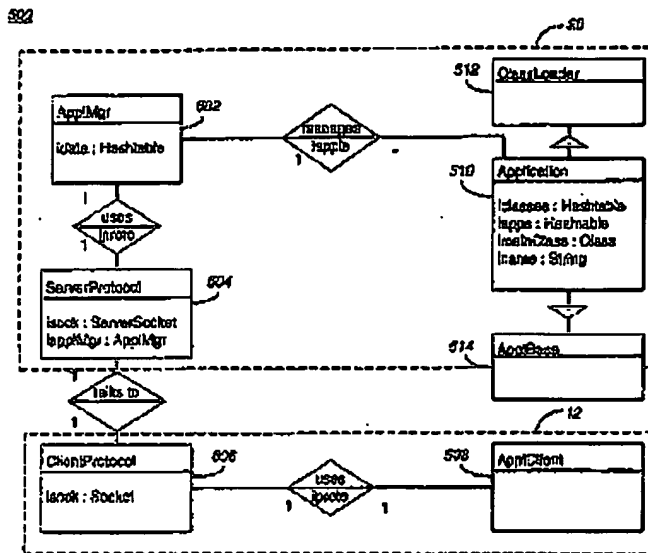
【図3】



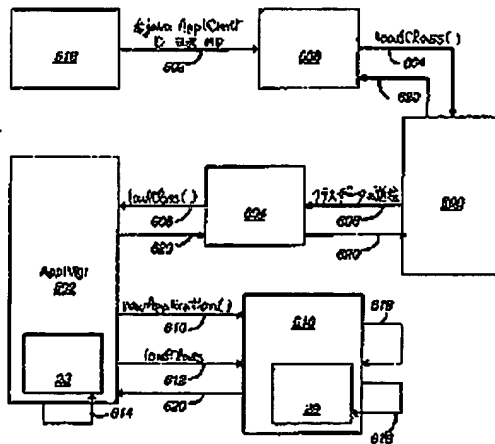
【図4】



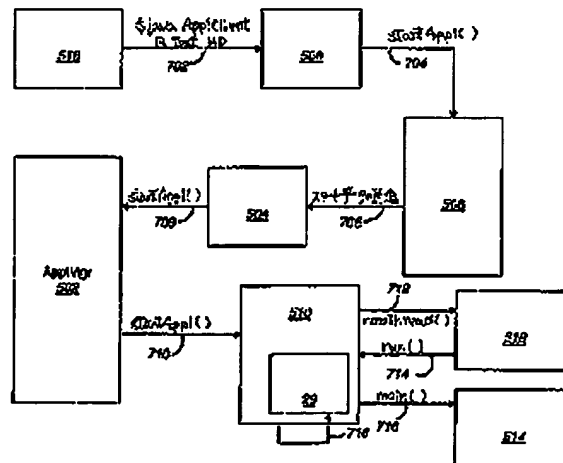
【図5】



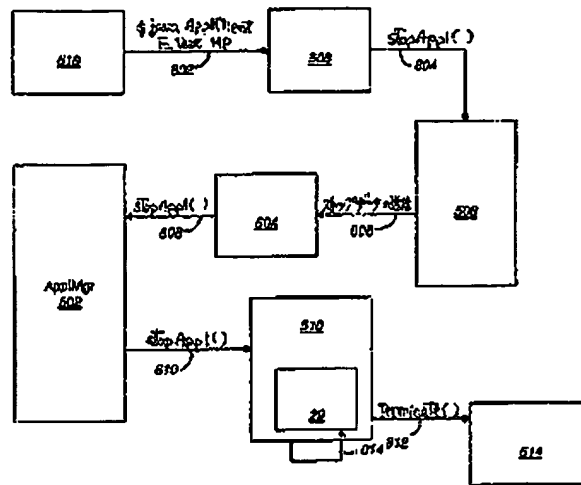
【図6】



【図7】



【図8】



【図9】

